MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. MEMO 459 JANUARY 1978

# PROGRAMMING

# VIEWED AS AN ENGINEERING ACTIVITY

Charles Rich, Howard E. Shrobe, Richard C. Waters
Gerald J. Sussman and Carl E. Hewitt

It is profitable to view the process of writing programs as an engineering activity. A program is a deliberately contrived mechanism constructed from parts whose behaviors are combined to produce the behavior of the whole. We propose to develop a notion of understanding a program which is analogous to similar notions in other engineering subjects. Understanding is a rich notion in engineering domains. It includes the ability to identify the parts of a mechanism and assign a purpose to each part. Understanding also entails being able to explain to someone how a mechanism works and rationalize its behavior under unusual circumstances.

Part of our methodology for investigating these ideas is to build a computer-aided design tool for computer programs. The construction of this tool will serve both as a concrete realization of our theoretical ideas and as a testbed for our practical techniques. We have in mind an interactive system used by an expert programmer to aid in maintaining the consistency of his design and performing routine tasks of analysis, synthesis and debugging. Our system will be able to explain the workings of the programs it understands. This is a more modest goal than trying to build a system which is itself an expert programmer. Nevertheless, the general availability of an interactive design aid such as we propose would significantly improve the quality of programs that are written. In addition, a program which understands other programs is a crucial first step towards programs which understand themselves and are therefore accountable for their own behavior.

This paper was adapted from a proposal to the National Science Foundation.

The research described in this paper was done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

# PROGRAMMING
# VIEWED AS AN ENGINEERING ACTIVITY

Charles Rich, Howard E. Shrobe, Richard C. Waters

Gerald J. Sussman and Carl E. Hewitt

It is profitable to view the process of writing programs as an engineering activity. A program is a deliberately contrived mechanism constructed from parts whose behaviors are combined to produce the behavior of the whole. We propose to develop a notion of understanding a program which is analogous to similar notions in other engineering subjects. Understanding is a rich notion in other engineering domains. It includes the ability to identify the parts of a mechanism and assign a purpose to each part. Understanding also entails being able to explain to someone how a mechanism works and rationalize its behavior under unusual circumstances.

How can program verification techniques be developed into a form of engineering analysis? How can engineering ideas such as modelling and equivalence be applied to the problems of software development? The time is ripe for a theoretical synthesis of several families of ideas; technical ideas from computer science can be combined with organizational ideas from engineering mediated by the methods of of artificial intelligence. A crucial step is the evolution of a computer science concept analogous to the engineering concept of a design plan -- a representation of the teleological relationship of the structure of a mechanism to its function.

Part of our methodology for investigating these ideas is to build a computer-aided design tool for computer programs. The construction of this tool will serve both as a concrete realization of our theoretical ideas and as a testbed for our practical techniques. We have in mind an interactive system used by an expert programmer to aid in maintaining the consistency of his design and performing routine tasks of analysis, synthesis and debugging. Our system will be able to explain the workings of the programs it understands. This is a more modest goal than trying to build a system which is itself an expert programmer. Nevertheless, the general availability of an interactive design aid such as we propose would significantly improve the quality of programs that are written. In addition, a program which understands other programs is a crucial first step towards programs which understand themselves and are therefore accountable for their own

behavior.

This paper develops the notion of plans as a general framework for understanding how programs are constructed. In developing this idea, we draw liberally upon our own intuitions and introspections about the activity of programming. We will show several examples of programs explained in terms of plans and discuss how plans can be formally represented. Next, we will describe our proposed computer-aided design system. Computer aided design is an established methodology in other engineering disciplines. We will adopt and extend this methodology in the domain of programming. Finally, we will try to place our research in the context of our previous work and the work of other researchers.

## The Engineering Context

An engineer applies knowledge from mathematics and physics to construct techniques and mechanisms which satisfy given desiderata. A typical engineering problem can be idealized as follows: given a starting set of devices with known behavior and set of rules by which these may be combined to produce more complex entities, construct a composite mechanism whose behavior satisfies certain specified properties. Engineering design is a set of methods engineers bring to bear on such problems.

For each design problem an engineer must establish the form of the answer. If the problem is of a familiar kind, he may retrieve several candidate forms. In most engineering domains the form of the answer is the description of the desired mechanism as a set of parts and their interconnections. In general this description has many undetermined parameters. The engineer's problem is then to determine if it is possible to instantiate one of these general answer forms according to the constraints of the particular problem. If a design problem is not familiar, it must either be reformulated into a familiar problem or decomposed into a combination of more familiar problems. The composition of solutions to sub-problems may lead to unforseen interactions requiring debugging.

The central organizing structure of the engineering design method is a plan which describes the mechanism being designed at many levels of detail. At each level there is a blueprint describing the interconnection of parts at that level of description. During the design process, information flows through the plan in several directions: up and down between levels of detail, and between parts on the same level of detail.

Synthesis moves information down in the plan. In synthesis the descriptions of parts of a composite mechanism are refined based on the description of the whole and the way the parts are interconnected to form that whole. This further specifies the properties of a part so that it can itself be designed.

Analysis moves information up in the plan by determining aspects of the behavior of the composite mechanism from appropriate aspects of the behaviors of its parts and their interconnection. When the composite object is itself part of a larger plan, this information constrains the design of neighboring parts in the larger plan.

At the end of the design process, the plan includes not only a description of the physical connections between the parts of a mechanism, but also a description of how the behaviors of the parts interact and constrain each other to produce the overall desired behavior.

Retrieving the form of the answer to familiar problems, synthesis, and analysis constitute the routine part of engineering design. Reformulating and decomposing unfamiliar design problems and debugging their solutions is a more creative part of the design process. We will restrict our initial investigations to the routine aspects of engineering design in the domain of computer programming.

## Plans and Teleology

The behavior of a device or part of a device can be described in two ways.

Some properties of a device are independent of its context of use. These properties constitute the intrinsic description of the device. For example, a capacitor can be described by the relation $i(t) = C\,dv(t)/dt$. Part of a complex mechanical assembly may be described as a "rod" or a "shaft". The LISP function APPEND can be described intrinsically by its input-output behavior of returning the concatenation of its arguments. Intrinsic descriptions correspond to specifications in the literature of software engineering.

A device may also be described by its role in the plan for a larger mechanism. This is its extrinsic description, or teleology. For example, a particular capacitor may be described as a coupling capacitor, a bypass capacitor, or a tuning capacitor, depending upon its purpose in the circuit. A purpose of a piston rod is to couple power from the piston to the crank shaft, while a purpose of a valve rod is to transmit control information from the

camshaft to the valve. Similarly, APPEND may be used to produce the union of two disjoint sets represented as lists, or to attach a suffix to a root word represented as lists of characters. The abstract form of the answer retrieved in the engineering design method is a plan in which each part is specified only by its extrinsic properties. Synthesis involves filling each role in the plan with a part whose intrinsic description satisfies the given extrinsic description.

A single part may have several extrinsic descriptions corresponding to multiple needs that it satisfies in the larger mechanism. For example, a screw in a camera may fasten two plates together and also provide a fulcrum about which to pivot a lever. There may also be several plans for a given device, describing its structure in different dimensions. In this situation, each part has the potential for one or more roles in each plan. For example, in a radio-frequency amplifier an inductor may be both part of a resonant circuit in the frequency domain plan and part of the bias network of a transistor in the DC plan.

The essence of understanding a mechanism is knowing the purposes of each part. This involves building a description of the mechanism which matches each part with its roles in the appropriate plans. Each role in each plan must be filled by some part of the mechanism and the intrinsic properties of that part must satisfy the extrinsic properties of its roles.

The utility of this kind of understanding is that it factors knowledge. Many devices share the same plan. Therefore understanding the teleological structure of a plan (which may be very difficult) need only be done once. It need not be repeated for each device whose principles of operation are based on that plan. If we prove certain properties of a plan, we know these properties will hold for all instances of that plan.

## Limits of Engineering

One major limitation faced by all physical engineering disciplines is in the accuracy with which the elements of the domain can be modeled. This is obvious in Civil Engineering where the actual properties of the engineering materials such as soil and concrete are only marginally understood. In Mechanical Engineering there is much better control of the properties of the materials, but many important processes such as wear, lubrication, and vibration are still inadequately modeled. Electrical Engineering has very accurate models for some of its basic components, as for example the Ebers-Moll model of a transistor. Unfortunately, very accurate models introduce a new kind of difficulty -- the

equations resulting from the use of such models are too complex to be solved. Circuit engineers are usually forced to construct a linear "small signal" model around some approximate operating point, which will hopefully be accurate enough for the analysis at hand. Computer science has the same difficulty with precise but unuseable models. For example, most computers have a floating point add instruction whose properties are precisely defined in the hardware manual for that machine. However, the existing methodology for analyzing programs which use these instructions usually requires intractably complex manipulations. As in Electrical Engineering, we are usually forced to assume a simpler model which only approximates the true behavior of the component.

A complexity problem also arises due to the interactions among components in an engineered system, even if the components are simple in themselves. In LISP, for example, the RPLACD operation has a simple description: it replaces the right half of a list cell with a new value. However, if this operation is used in a program which has many linked lists with shared structure, its effect can be very complicated to describe.

Thus complexity arises in two ways. The parts of a system may be hard to describe precisely, and the interactions among the parts may be complicated. In both these cases we produce composite objects whose descriptions are practically unuseable.

How do we keep complexity under control? In large social organizations, complexity is controlled by establishment of bureaucratic structures which separate parts of the system and enforce stylized means of communication among parts which interact. Engineered systems contain similar bureaucracies. One of our goals is to understand the techniques of building and managing these bureaucracies.

One traditional bureaucratic organization is <u>hierarchy</u>. Groups of electronic components are organized into amplifiers, oscillators, gates, and power supplies. If an amplifier can be built whose properties are simply and accurately describable, it is irrelevant that the descriptions of the transistors it contains are complex or inaccurate. Computer science also employs hierarchy to advantage. Programs are organized by means of block structure, subroutines and hierarchies of data. The description of a subroutine is often much simpler than the description of its parts.

A related notion is abstraction, the throwing away of detail which is not relevant to the task at hand. For example, expert circuit designers make use of terminal and port equivalences for summarizing the behavior of parts of a circuit. A similar technique is frequently used by programmers, although it is rarely talked about in terms of formal equivalences. For example, when trying to debug one part of a program, it is common to model the behavior of other parts of the programming environment in a very general way (e.g. "that part of the program searches the data base") which hides a myriad of currently irrelevant details. Depending on the question being asked, different sets of details are relevant. This is why there may be more than one plan for a device.

## Engineering and Macroscopic Reasoning

Much of physical reality is, in principle, described to a high degree of accuracy by the theory of quantum electrodynamics. Given this fact, one might propose to use this theory directly, for example to explain biological phenomena by reduction to Schroedinger's equation. Practical experience suggests that little success can be expected along this route since the complexity of the resulting explanation turns out to be intractable.

Similarly, in programming the connection between the microscopic and the macroscopic cannot be direct. Program analysis which concentrates on the axiomatic description of program primitives is inadequate for dealing with the complexity of real world programs. Instead, one needs to develop abstractions appropriate to the task at hand. Without these, the resulting description is too complex to be manipulated or understood.

Consider applying a microscopic theory to a program as simple as a merge of two ordered lists. The program works by splicing elements from the first list into the second in such a way that the resulting list is ordered. The microscopic approach begins by describing the initial state of every cell in the computer memory. Important properties of the lists, such as being ordered, are derived from the states of the individual cells before and after each side-effect operation. Manipulating the resulting "equations" is infeasible, and even if we had a machine fast enough to handle these computations, the result would be incomprehensible.

An engineering approach works with higher level notions. It is more fruitful to think about two lists and splicing as a kind of operation on these higher level objects, rather than thinking of computer memory as a large collection of cells and about changing pointers in particular cells. In this way, we are much more likely to arrive at a computationally feasible and easily understandable description of the behavior of such a program.

The higher level notions used in the engineering approach can be related to the primitive level of description. They are not a different theory of phenomena but rather a means of organizing the basic principles into useful structures which allow us to calculate what we need to know. In our research, we intend to identify the higher level notions that allow expert programmers to build and understand real-world programs. We will do this by developing a formal system and a programming environment in which these abstractions can be embedded and tested. The abstractions will be evaluated by their ability to help a computer program analyze and understand programs.

## An Example

The following example illustrates some of the methods we bring to bear on the problem of understanding programs. Specifically, we display how knowledge of the plan for a program constrains and directs the analytic process.

A basic programming plan is the enumeration loop, which enumerates a sequence of values. A variety of more complex loops can be constructed from this plan by adding other features. For example, a filter can be added in order to select some subset of the sequence of values for special processing. Another important plan which may be combined with an enumeration loop is the accumulation, in which some aspect of a number of objects are combined using an associative operator (such as sum or union).

The program below illustrates the use of these programming techniques. Given an array encoding the pay status and salary of all employees, this program computes the total salary paid to those employees who are paid biweekly.

```
10    sum ← 0;
20    FOR i ← 1 STEP 1 UNTIL #employees DO
30        IF employee[i,status] = biweekly
40            THEN sum ← sum + employee[i,salary];
```

The plan for this program, which will be referred to as a filtered accumulation loop, consists of an enumeration loop, a filter to select those employees who are paid biweekly, and an accumulation which sums the salaries of those employees which pass the filter.

An enumeration loop has three parts: initialization, bump, and end-test. The most essential feature of an enumeration loop is the sequence of values it enumerates. Here the enumeration loop enumerates the employees represented by the integers 1 to *employees. The current element being enumerated is the value of i. The initialization, bump, and end-test of this enumeration loop are all expressed by the FOR construct (line 20): the initialization sets i to 1; the bump adds 1 to i; the end-test halts the loop after the last employee has been enumerated.

A filter is a predicate which is used to select a subset of the elements being enumerated. In the example, line 30 selects the values of i which represent employees whose pay status is biweekly.

An accumulation consists of three parts: initialization, the accumulation operator, and the contribution function. In the example, the accumulation operator is "+" (line 40). The initialization assigns 0 to sum (line 10). The contribution function (which computes the contribution of the current element to the accumulation) computes the employee's salary from the current employee number i.

How do we know that the above program computes the total salary paid to biweekly employees? We assume that the employee array contains information on all of the relevant employees, and that the first coordinate of this array runs from 1 to *employees. Given this, we know that the enumeration loop enumerates all of the employees; the filter selects just those which are paid biweekly; and the accumulation sums their salaries.

The important thing to notice here is that the form of this correctness argument (this is not yet a formal proof, but it could be made into one) follows from the form of the plan. Specific parts of the program fit into the argument in a way which is determined by the part of the plan which they implement.

The power of using plans has been to factor the problem of understanding how a program works into three more tractable sub-problems: recognizing the underlying plan of the program; establishing how each part of the plan is implemented by the code; and developing an understanding of the underlying plan. This factorization has the additional advantage that the number of fundamentally different underlying plans is much smaller than the number of possible programs based on them. Thus once we have developed an understanding of a basic plan, such as filtered accumulation loop, this can be applied to many programs whose teleology may superficially appear to be totally different.

## A More Complicated Example

The following program computes the intersection c of the two sets a0 and b0, where each set is represented as an ordered list with the smallest element first.

```
10    a ← a0
20    b ← b0
30    c ← list();
40    WHILE ¬empty(a) ∧ ¬empty(b) DO
50          IF head(a) = head(b)
60             THEN BEGIN c ← append(c,list(head(a)));
70                        a ← tail(a);
80                        b ← tail(b)
90                  END
100            ELSE IF head(a) < head(b)
110                    THEN  a ← tail(a)
120                    ELSE  b ← tail(b)
```

This program can be understood by recognizing that its underlying plan is also a filtered accumulation loop. Analysis of the data flow in the program shows that line 60 is an accumulation which is initialized by line 30. It also shows that the rest of the program forms an enumeration loop. The fact that the accumulation step is not executed on every iteration of the loop, but only when the predicate on line 50 is satisfied, indicates that line 50 is a filter. Once a correspondence has been established between the parts of this program and the parts of the abstract plan for a filtered accumulation loop, the plan can be used as a guide for understanding how the program works.

The essential feature of an enumeration loop is the sequence of values it enumerates. Here elements of the union of a0 and b0 are enumerated in order starting with the smallest. The variables a and b hold the elements of the sets a0 and b0, respectively, which have not yet been enumerated. On each iteration of the loop, the bump (lines 50 and 70-120) selects the smallest element not yet enumerated and then removes it from a or b, or both. To achieve this the program takes advantage of the fact that the lists a and b are ordered and therefore the smallest element must be the first element of a or b, or both. Notice that in this way of coding the program there is no variable which has the currently enumerated element as its value. In lines 60-80 and 110 the enumerated element is head(a) and in lines 60-80 and 120 it is head(b).

The filter (line 50) selects the enumerated elements which are elements of both a0 and b0. These are the elements of the intersection of a0 and b0. The accumulation (line 60) produces an ordered set of these elements.

The use of append as the accumulation operator is an interesting example of the difference between intrinsic and extrinsic behavior. Intrinsically, append(c,list(head(a))) concatenates head(a) onto the end of list c. In this particular situation, however, this achieves the operation of adding head(a) as a new element of the set represented as an ordered list in c. This is possible because the enumeration loop in which this accumulation is embedded enumerates elements in order and without duplicates.

The filter (line 50) is another example of the difference between intrinsic and extrinsic behavioral descriptions. Intrinsically line 50 checks whether or not the first elements of a and b are identical. Extrinsically, it checks whether or not the currently enumerated element is in the intersection of a0 and b0. It is clear that if the identity test succeeds, the enumerated element is in the intersection. The fact that when the test fails the enumerated element is not in the intersection, is not so obvious. There are two cases: first suppose head(a) < head(b), then the enumerated element is head(a) and it cannot be an element of b0 because it is less than all the elements of b and greater than all the other elements of b0 (those which have already been enumerated); alternatively, if head(b) < head(a) the argument is the same with the roles of a and b reversed.

Notice that line 50 fills two roles in the plan for this program. It is the filter, and it is also part of the bump for the enumeration loop. This sharing makes the program more compact, but less clear. Using plans allows us to separately identify the multiple roles of a piece of code in order to understand how the program works.

A final complexity in this example program arises from the fact that the end-test of the enumeration loop (line 40) may stop the loop before all of the elements of the union of a0 and b0 are enumerated. However, when it stops the loop one of the sets a or b must be empty. Thus the remaining elements cannot be members of both a0 and b0.

The proof of correctness for this program has the same structure as the proof for the first example program. First we have to show that the enumeration loop enumerates all of the elements of a0 and b0 which could possibly be in the intersection. Then we must show that the filter selects only the elements which are in the intersection. Finally, we must show that the accumulation constructs an ordered set of the selected elements.

The intersection program has been understood in terms of the same simple plan as the first example program. The greater complexity of this example is due to the fact that the sequence of elements being enumerated is less obvious, and that the code has been optimized by sharing (line 50) and taking advantage of the properties of ordered lists.

## A Different Example

This example will bring out two points. First we introduce several new plans (driver loop, transitive closure and demons) which operate at a higher level of abstraction than plans such as filtered accumulation loop. These plans are defined more in terms of teleological structure than patterns of data and control flow. However these plans play the same role in this example as the filtered accumulation loop plan in the previous examples; they are representations of programming knowledge and are used to develop the form of a proof of correctness.

The second point illustrated by this example is the degree to which plans may be nested, allowing a program to be understood at many levels of detail. In this example a very abstract plan (transitive closure) is implemented by a more specific plan (demons) which is in turn implemented using queues. Furthermore, the function calls in this example hide large and complicated sections of program, such as pattern matching and data base retrieval.

The following program is the top-level loop of a procedural deduction system like those used in artificial intelligence systems. The program reads in facts and asserts them in a data base. As each fact is asserted, it triggers applicable demons. The effect of processing these demons is to create new facts, which are called the repercussions of the original fact. These repercussions must also be asserted in the data base, which may trigger additional

demons which assert additional repercussions, and so on. The central claim of the specifications for this program is that all of the repercussions of every asserted fact will eventually be asserted.

```
 10 PROCEDURE run (user-stream); BEGIN
 20 fact-queue ← new-queue();
 30 demon-queue ← new-queue();
 40 data-base ← create-new-table();
 50 WHILE true DO BEGIN
 60       fact-queue ← enqueue(read(user-stream), fact-queue);
 70       WHILE ¬empty(fact-queue) DO BEGIN
 80             assert-new-fact(first(fact-queue), data-base);
 90             demon-queue ← trigger-demons(first(fact-queue), data-base)
100             fact-queue ← dequeue(fact-queue);
110             WHILE ¬empty(demon-queue) DO BEGIN
120                   new-facts ← process-demon(first(demon-queue));
130                   fact-queue ← append(fact-queue, new-facts);
140                   demon-queue ← dequeue(demon-queue)
150                   END
160             END
170       END
180 END
```

At the topmost level of description, the plan for this program is a composition of the driver loop plan and the transitive closure plan. The driver loop is a simple non-terminating loop which reads in data from a user and then performs calculations based on this data. The READ-EVAL-PRINT loop of LISP is an example of a driver loop.

Lines 70-160, which perform the calculation part of the driver loop, can be understood in terms of the transitive closure plan. Like any plan, the transitive closure plan specifies how sub-segments can be combined in order to achieve a particular purpose. Here the purpose is to compute a transitive closure, specifically: given an operator, the current closure, and a new item to be added to the closure, the plan gives one way of computing an enlarged closure which contains the new item and enough other new items so that it is still closed under the operator. (Here the operator is the computation of repercussions by the demons). The structure of this plan is based on a lemma which states that transitive closure is accomplished given that two conditions are satisfied: (i) whenever an item is added to the

closure, all of the images of that item under the operator are eventually calculated; (ii) every item which is calculated will eventually be added to the closure.

The transitive closure plan is very abstract, allowing for many specific implementations. This program uses demons. Demon plans are characterized by the following features: there are two data bases, one containing facts, and the other containing demons; there is a method of triggering the demons that are applicable to a given fact; and the execution of triggered demons results in the creation of new facts. The teleological structure of demon plans is expressed by the following four lemmas:

(i) For any fact asserted in the data base, all applicable demons are eventually triggered.
(ii) Every triggered demon is eventually executed.
(iii) Execution of the demons triggered by a given fact produces the immediate repercussions of that fact.
(iv) All facts produced by demon execution are eventually asserted in the data base.

Lemmas (i)-(iii) above implement condition (i) of the transitive closure plan. Lemma (iv) above implements condition (ii) required for transitive closure

The demon plan itself is still abstract enough to allow considerable variation in the way it is implemented. Most of this variability comes from the ways in which "eventually" can be implemented. In the above program, two queues are used. One contains facts which are waiting to be asserted in the data base. The other contains demons which are waiting to execute and produce the repercussions of those facts which have been asserted. Still lower level plans describe how these queues are used to implement the lemmas of the demon plan. In brief, the loop which empties the fact-queue (lines 70-160) satisfies lemmas (i) and (iv) of the demon plan, while the loop which empties the demon-queue (lines 110-150) satisfies (ii) and (iii).

Thus the logical structure of this program is embedded in three levels of plans, each satisfying the requirements of the next higher level of structure. Below these plans lie yet more layers of plans and abstractions which implement the queues and the data bases. The program's analysis is factored into manageable steps by this hierarchy of plans and abstractions.

## Representing Plans

In order to formalize the kind of program understanding demonstrated in the examples above, we have developed a representation for plans and investigated the use of this representation to describe various types of programs.

A plan is a hierarchical network of segments with three kinds of links between them: data flow, control flow, and purpose links. Each segment has specifications which are a formal statement of the conditions and relationships between objects which are expected to hold prior to and immediately following execution of the segment.

Data flow links specify how data objects flow between segments. Control flow links specify the order in which segments are executed. The teleological structure of a plan is expressed by purpose links, which relate the specifications of segments to one another. The two simplest types of purpose links are prerequisite links, which represent the fact that the input expectations of one segment are satisfied by the output assertions of another segment, and achieve links, which record how the output assertions of a sub-segment figure into achieving the output assertions of the segment of which it is a part.

Figure 1 shows the data flow and control flow in a plan for a filtered accumulation loop. It is neither the most general possible plan for filtered accumulation loops, nor the plan of any particular program -- it is rather the kind of intermediate level plan that is the common knowledge of every practicing programmer. The names of segments are in upper case, data objects are in lower case. Solid lines denote data flow. Dotted lines are control flow links.

Specifications for segments in the plan are given in Table 1a and 1b following the figure. A specification has four parts: a list of input objects (INPUTS:), a list of output objects (OUTPUTS:), pre-conditions (EXPECT:), and post-conditions (ASSERT:). These pre- and post-conditions are are written in a formalism developed by Rich and Shrobe and described in detail in [Rich & Shrobe, 1976]. Briefly, each clause in an EXPECT or an ASSERT is a predicate on the data objects, written in prefix notation. Square brackets denote functional terms. Thus if (FIRST SEQUENCE1 OBJECT1) is the predicate meaning OBJECT1 is the first part of SEQUENCE1, then [FIRST SEQUENCE1] denotes "the object which is" the first part of SEQUENCE1

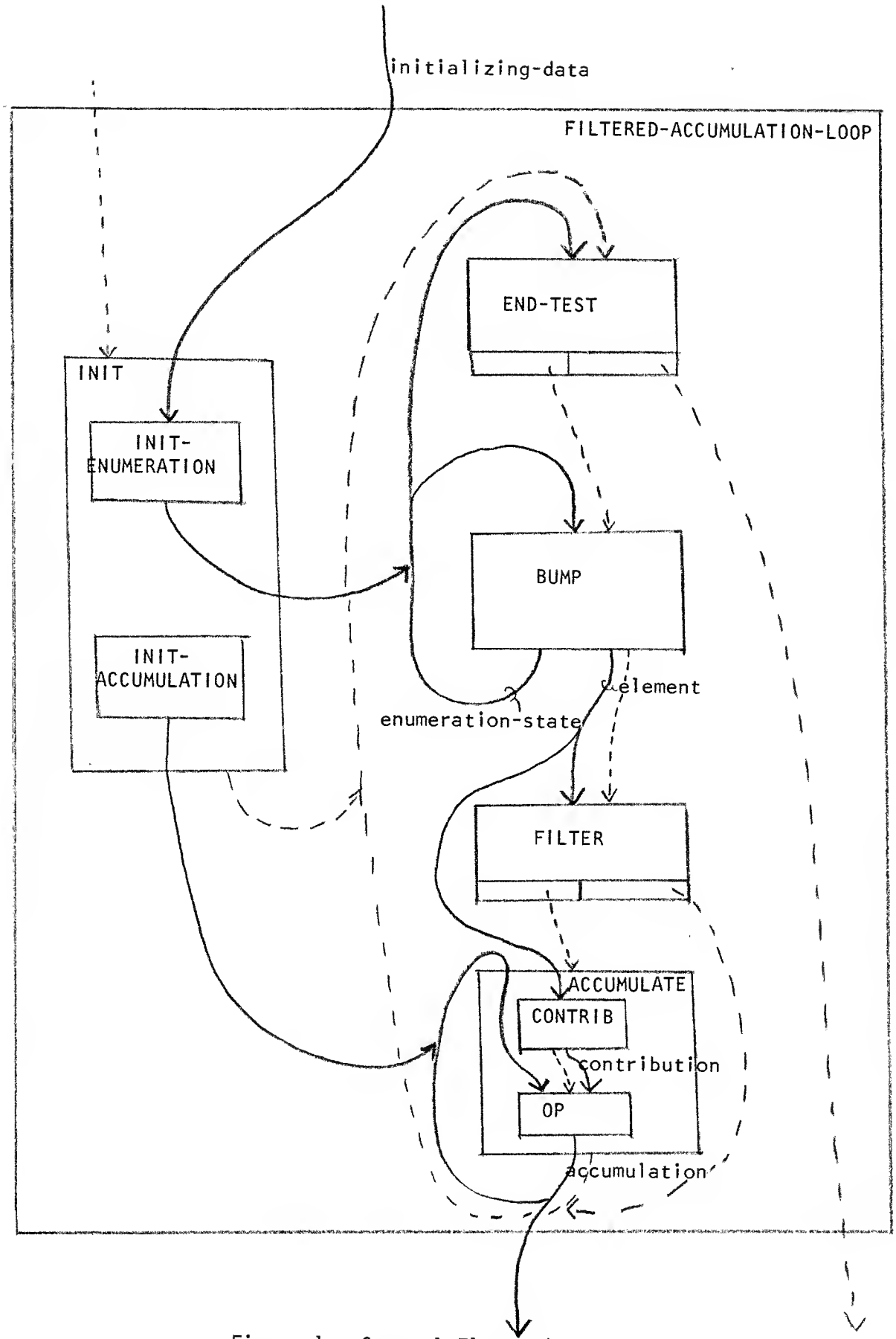


Figure 1. Control Flow and Data Flow in Plan for Filtered Accumulation Loop

```
(DEFSPECS FILTERED-ACCUMULATION-LOOP
    (INPUTS: INITIALIZING-DATA)
    (EXPECT:)
    (OUTPUTS: ACCUMULATION)
    (ASSERT: (PI ASSOC-OPERATOR
                 (ELEMENT-OF RESTRICTED-SEQUENCE =X) [CONTRIBUTION-OF =X]
                 ACCUMULATION)))


(DEFSPECS INIT-ENUMERATION
    (INPUTS: INITIALIZING-DATA)
    (EXPECT:)
    (OUTPUTS: ENUMERATION-STATE)
    (ASSERT: (CURRENT-ELEMENT-OF ENUMERATION-STATE [FIRST SEQUENCE])))


(DEFSPECS END-TEST
    (INPUTS: ENUMERATION-STATE)
    (EXPECT:)
    (OUTPUTS:)
   (CASE1 (ASSERT: (EXHAUSTED ENUMERATION-STATE)))
   (CASE2 (ASSERT: (NOT (EXHAUSTED ENUMERATION-STATE)))))

(DEFSPECS BUMP
    (INPUTS: ENUMERATION-STATE1)
    (EXPECT: (NOT (EXHAUSTED ENUMERATION-STATE1)))
    (OUTPUTS: ELEMENT ENUMERATION-STATE2)
    (ASSERT: (CURRENT-ELEMENT-OF ENUMERATION-STATE1 ELEMENT)
             (NEXT-STATE ENUMERATION-STATE1 ENUMERATION-STATE2)))
```

**Table 1a. Specs for Segments in Filtered Accumulation Loop Plan.**

```
(DEFSPECS FILTER
   (INPUTS: ELEMENT)
   (EXPECT: (MEMBER SEQUENCE ELEMENT))
   (OUTPUT:)
  (CASE1 (ASSERT: (MEMBER RESTRICTED-SEQUENCE ELEMENT)))
  (CASE2 (ASSERT: (NOT (MEMBER RESTRICTED-SEQUENCE ELEMENT)))))


(DEFSPECS INIT-ACCUMULATION
  (INPUTS:)
  (EXPECT:)
  (OUTPUTS: ACCUMULATION)
  (ASSERT: (ZERO-OF ASSOC-OPERATOR ACCUMULATION))


(DEFSPECS CONTRIB
  (INPUTS: ELEMENT)
  (EXPECT:)
  (OUTPUT: CONTRIBUTION)
  (ASSERT: (CONTRIBUTION-OF ELEMENT CONTRIBUTION)))


(DEFSPECS OP
   (INPUTS: CONTRIBUTION ACCUMULATION1)
   (EXPECT:)
   (OUTPUTS: ACCUMULATION2)
   (ASSERT: (ASSOC-OPERATOR CONTRIBUTION ACCUMULATION1 ACCUMULATION2)))
```

**Table 1b. Specs for Segments in Filtered Accumulation Loop Plan (cont'd).**

Underlined symbols in specifications are uninterpreted at the current level of abstraction. These symbols are instantiated (within specified constraints) when an abstract plan is applied to a particular program. For example, the plan for a filtered accumulation loop at this level of abstraction is not committed to what particular sequence is being enumerated, which restricted sub-sequence is being selected by the filter, what the associative operator is used to form the accumulation, or what is the contribution of each selected element. Additional input objects may also be added to these specifications when they are applied to a particular program. For example, the filter segment often compares the current element with some external input object.

The first specification in Table Ia is a statement of the overall behavior of a filtered accumulation loop. PI denotes the composition of a given function over a sequence (if the function is addition this would be the usual algebraic SIGMA notation). Thus the specification for FILTERED-ACCUMULATION-LOOP asserts that the output object is the composition, using a given associative operator, over all elements of a restricted sequence, using a given contribution function for each element.

To decompose the teleological structure of the filtered accumulation loop, we consider the way the plan is built up. The "backbone" of the plan is an enumeration loop, which has three sub-segments, the initialization, the bump, and end-test. Specifications for these segments are shown in Table Ia. The importance of recognizing these three sub-segments as a separate grouping in the plan is that there are simple standard lemmas and methods of proving things about enumeration loops. The specifications of these sub-segments are also fairly abstract. The data object called the enumeration state represents some way of encoding the current element being enumerated, and a way of getting to the next element in the sequence. In the first example program, the enumeration state was implemented simply as the current index (i) in the array. In the second example, the current values of a and b encoded the state of the enumeration.

Filtering may be added to an enumeration loop by inserting a FILTER segment at the appropriate place in the control flow and data flow. As shown in Figure 1, the new segment is inserted in the control flow between the BUMP and the END-TEST. Its only connection with the data flow of the enumeration loop is that it uses the ELEMENT output of the BUMP. This also alters the logical structure of the plan. Associated with filtering is knowledge about how to prove the new specifications. In order to prove that a filter selects a particular restricted sequence, first show that the restricted sequence is a subset of the enumerated sequence, and then show that the filter selects exactly those elements which are in the restricted sequence.

Another method of building a more useful plan out of an enumeration loop is to add accumulation. Figure 1 shows how the various sub-segments involved in accumulation fit into the rest of the plan. The initialization is put with the initialization of the enumeration, while the contribution function and the accumulation step are put in the control environment created by CASE1 of the filter. The only connection with the data flow of the surrounding plan is again the use of the current ELEMENT.

The accumulation part of the plan carries with it an important lemma: if the accumulation is initialized to the zero of the operator, and the operator is associative, then it computes the PI of that operator on the sequence of elements given as input to the ACCUMULATE segment.

This view of building up the filtered accumulation loop from enumeration, filtering, and accumulation suggests how to understand particular programs based on this plan: first discover what sequence the enumeration loop enumerates and show that the restricted sequence is a subset of it; second, show that the filter performs the proper restriction; finally, use the lemma about accumulations to conclude that the plan computes the required combination of contributions.

Now we can consider the purpose links in this plan. Each EXPECT of a sub-segment needs to be satisfied by the ASSERT of some other sub-segment (or the EXPECT of the overall segment, though this does not occur in this example). Only two sub-segments have EXPECT's in this abstract plan.

(i) BUMP; (EXPECT: (NOT (EXHAUSTED ENUMERATION-STATE1)))
which depends on
END-TEST: (CASE1 (ASSERT: (NOT (EXHAUSTED ENUMERATION-STATE)))

(ii) FILTER: (EXPECT: (MEMBER SEQUENCE ELEMENT))
which depends on
BUMP: (ASSERT: (NEXT-STATE ENUMERATION-STATE1 ENUMERATION-STATE2))

The first of these prerequisite links is straightforward. The second link involves a step of deduction from the meaning of the predicate NEXT-STATE and the data flow connection between BUMP and FILTER in the plan. The NEXT-STATE operation corresponds to moving from one element of a sequence (the current element of the old state) to the next element of that sequence (the current element of the new state).

The FILTERED-ACCUMULATION-LOOP segment has only a single output assertion, (PI ;..), which depends on the output assertions of all the sub-segments. The logical structure of this achieve link has been explained in detail for each of the two example programs. It involves decomposing the operation of the plan into enumeration, filtering, and accumulation and making use of standard lemmas about each of these parts.

## Relating Abstract Plans to Concrete Programs

Plans are more abstract than programs in two ways. First, plans are a procedural abstraction. As we have seen in the examples, understanding the underlying plan of a program entails abstracting away from the syntactic "sugaring", such as FOR or DO WHILE or GOTO, that is used to achieve the ordering of segments in control flow. Similarly, the data flow is abstracted from the particular syntactic techniques used to achieve it, such as the use of variables, return values, and so on, depending on the particular programming language. Plans also include data abstraction. The plan for a filtered accumulation loop is written in terms of abstract data types such as "sequence" and "accumulation". The unification of procedural and data abstraction is one of the most novel and important properties of our system of description.

The precise relationship between a concrete program and the more abstract underlying plan needs to be explicitly represented and reasoned about. For example, in the employee enumeration program the abstract data object ENUMERATION-STATE is implemented in three parts: an array, and two indices. The part of the sequence yet to be enumerated is the array section between the two indices. Table 2 shows the notation we are developing to represent this kind of knowledge. The first form in the table names the abstract object being implemented and gives the name ARRAYSECTION to this particular implementation scheme. The next section names the three more concrete objects which are used in this implementation: ARRAYPART, BOTTOM-INDEX, and TOP-INDEX. Finally, the IMPLEMENTATION-DEFINITION's describe how each abstract operation on the enumeration state is implemented in terms of the more concrete objects.

Of course this only does part of the job. What is not shown here is the mapping of abstract data objects all the way down to the particular variables used in the program (for example that the BOTTOM-INDEX is i and the TOP-INDEX is #employees). This also applies to the data flow and control flow. For each link in the plan diagram above, we record how that link is achieved in the code. We prefer to de-emphasize this part of the program understanding problem because it is very dependent on the particular programming

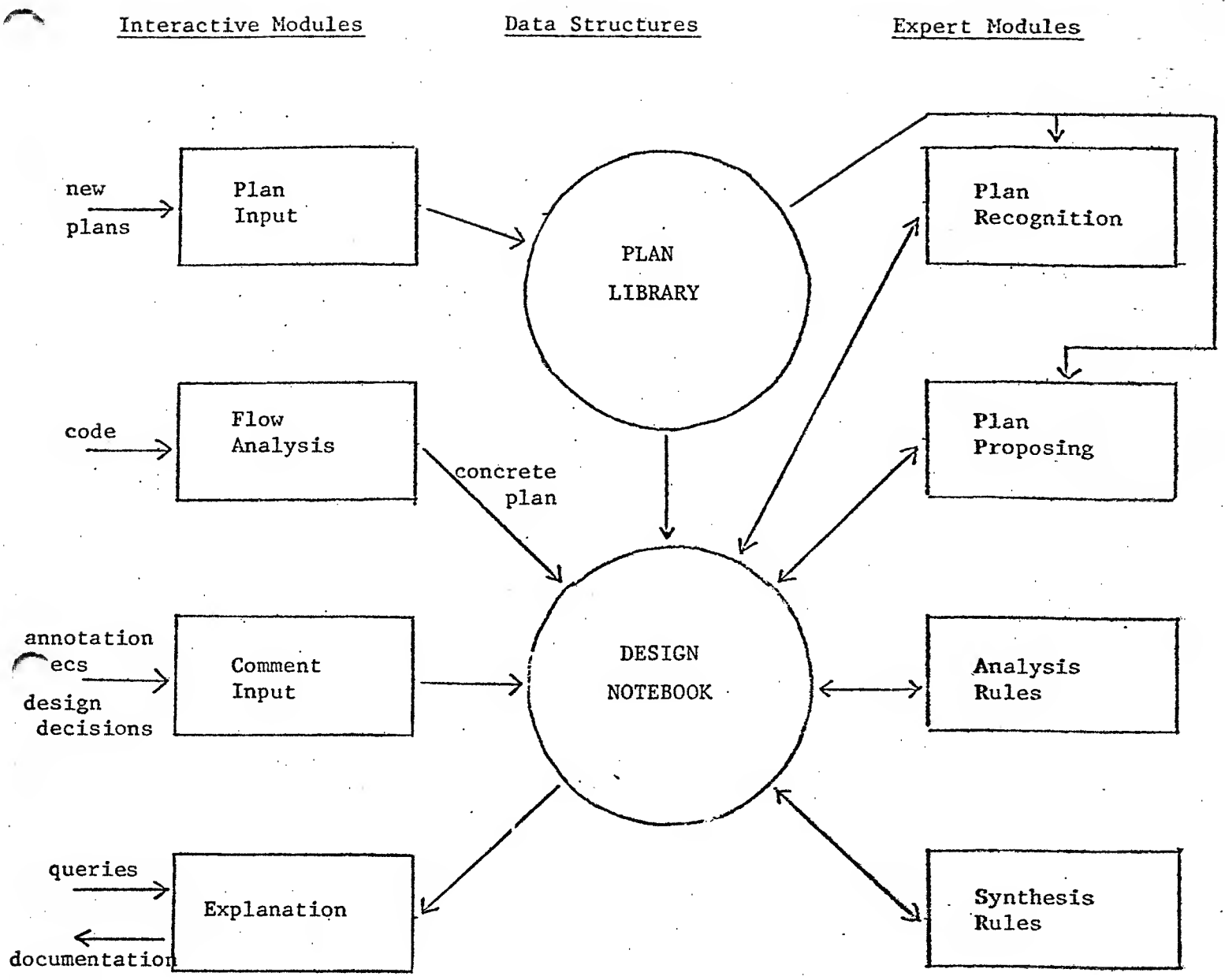


Figure 2. Structure of System.

language being used. We are primarily interested in representing structures that are universal, such as FILTERED-ACCUMULATION-LOOP or the above scheme for implementing an enumeration.

## A Library of Plans

There are many different plans in common programming use. It is helpful to organize these plans along two orthogonal dimensions: "concrete" to "abstract" and "simple" to "compound".

A given program can be described by plans at various levels of abstraction. A concrete plan is a pattern of control flow and data flow with some additional constraints. An abstract plan is a logical structure of goals and lemmas which expresses the underlying teleological structure of a program. For example, enumeration loop is a more concrete plan than transitive closure.

Enumeration loop and transitive closure are simple plans. Compound plans are larger structures built up by combining simple plans. For example, filtered accumulation loop is a compound plan built up from an enumeration loop, a filter, and an accumulation.

Control and data flow analysis of the code for a program yields the most concrete plan for that program, which for non-trivial programs is a compound plan. Deeper analysis yields plans for the same program at higher levels of abstraction. The plan at one level is implemented by the more concrete plan at the level below.

The concrete plan for a program factors out much of the surface detail of the program, so that there are fewer concrete plans than there are coded programs. For example, a program implemented using imperative iteration with assignment statements and an equivalent program implemented using tail recursion are both mapped to the same concrete plan. However, there is still a very large number of plans. Fortunately, it appears that compound plans are built up from a manageably small number of simple types of plans. Waters [Waters 77] has investigated this informally by making a manual analysis of 44 programs in the IBM Scientific Subroutine Package. He found that the seven simple concrete plans discussed below were sufficient for building the compound concrete plans for most of these programs. We also believe that compound abstract plans can be decomposed into a manageable number of simple abstract plans.

One of the goals of our research is to develop a library of useful plans. In general, very large compound plans do not appear in such a library because they have limited applicability. Considerable work has already been done on simple concrete plans.

The simplest concrete plans are conjunction, conditional and composition. These three plans correspond to three ways of decomposing a problem: it can be divided into a set of independent sub-problems (conjunction); it can be classified into sub-cases (conditional); or it can be divided into sub-problems whose solutions are cascaded (composition). Thus the conjunction plan combines a set of sub-segments which do not interact. Each sub-segment is executed once in order to achieve the overall effect required. The conditional plan combines a multiple-case predicate with a set of sub-segments. On any given execution of the plan, only one of the sub-segments is executed in order to achieve the overall effect. The composition plan combines two sub-segments such that the first has data flow into the second. The two segments are executed in sequence to achieve the overall effect.

Loops form another major class of simple concrete plans. The enumeration loop is the simplest plan of this class. In an augmented loop, a computation is performed on each iteration of the loop which is independent of the computations on other iterations. A program which copies a vector by copying the elements one at a time is an example of an augmented loop. An accumulation loop is more complicated than this, since there is a data object, the accumulation, which is passed between iterations. A final example from this class of plans is the interleaved loop, which combines two loops so that they are executed in synchrony. The combination terminates as soon as either one terminates. This plan is most commonly used to bound the execution time of a loop by interleaving it with another loop which is known to terminate.

Other concrete plans which need to be added to the above list include more complicated loop augmentations, plans based on interrupts and error exits, coroutining plans, and plans based on more complex data structures, such as trees. There are also many important abstract plans which need to be formalized, such as satisfying a set of constraints by relaxation, successive approximation, and operating like a finite state machine.

## Proposal for a Program Understanding System

As part of our research, we propose to implement and experiment with a computer system that can understand programs using the methods and formal representations presented above. This system will be the prototype of a computer-aided design tool for expert computer programmers called a programmer's apprentice [Rich & Shrobe, 1976] [Waters, 1976] [Smith & Hewitt, 1975]. A programmer's apprentice is an interactive programming environment in which both the computer and the human programmer cooperate to produce software more quickly and reliably than either could do working alone.

In the apprentice environment the programmer will treat the computer as if it were a colleague, explaining and developing the program design interactively. The apprentice however will play a substantially passive role; its task is not design but rather careful bookkeeping and criticism. If the apprentice does not understand what the programmer has told it, its job is to complain, forcing the programmer to either debug his current plan or to provide sufficient detail for the apprentice to understand him.

The apprentice will be able to provide the programmer with services such as maintaining the consistency of the design and explaining the program in high level terms convenient to the task at hand. These tasks depend on the apprentice's ability to analyze the programmer's design and code. In addition, the apprentice should be able to conduct routine synthesis tasks. For example, in designing a large system the programmer might invoke an unimplemented sub-segment in several different contexts. The apprentice should be able to combine the various requirements placed on this uncoded module into a module specification and search the plan library for a plan which might be able to satisfy these specifications.

The apprentice is not expected to be a program designer, nor is it required to be super-human at the jobs it is assigned. It will often be unable to complete many of the tasks it sets out to accomplish. However, it will always be able to report what it has learned in any attempt and what it has been unable to understand. If its knowledge base is rich enough, the apprentice will be able to interact smoothly with the programmer, providing far more assistance than annoyance.

The internal structure of the apprentice is shown in the Figure 2 which is separated into interactive components, expert components, and mediating data structures. The interactive components accept design statements, code, and teleological annotation from the programmer and provide him with program explanation, design criticism, and other assistance. At the right side of the diagram are the modules whose job it is to analyze, synthesize, recognize, and propose plans. These experts are implemented as rule-based systems capable of modular extension. Mediating between these modules are two main data structures, the design notebook and the plan library.

The design notebook records the apprentice's current knowledge and beliefs about the program being designed and serves as the communications center for the entire system. The notebook includes the actual code written thus far, the programmer's comments, a current "working plan" for the program, and teleological annotation explaining how the program implements this plan.

The various modules communicate with one another by making assertions in the notebook. Each module has predefined trigger patterns which cause it to perform specific tasks, such as making a deduction or querying the user, whenever appropriate assertions appear in the notebook. Every assertion added to the notebook is also accompanied by a justification for its presence (a justification of special importance is that the programmer said so). These justifications make it possible for the apprentice to account for its actions when required.

The assertions in the notebook are used to represent the following kinds of information:

(i) The sequence of increasingly refined plans which partially accomplish some of the tasks specified.

(ii) Partial specifications for some of the subtasks which are to be accomplished.

(iii) Partial justifications regarding how some of the plans satisfy some of their specifications.

(iv) Partial descriptions of some of the background knowledge (mathematical facts, physical laws, government regulations, etc.) of the environment in which the system will operate.

(v) A collection of scenarios (at various articulations of detail) demonstrating how the system is supposed to work in concrete instances.

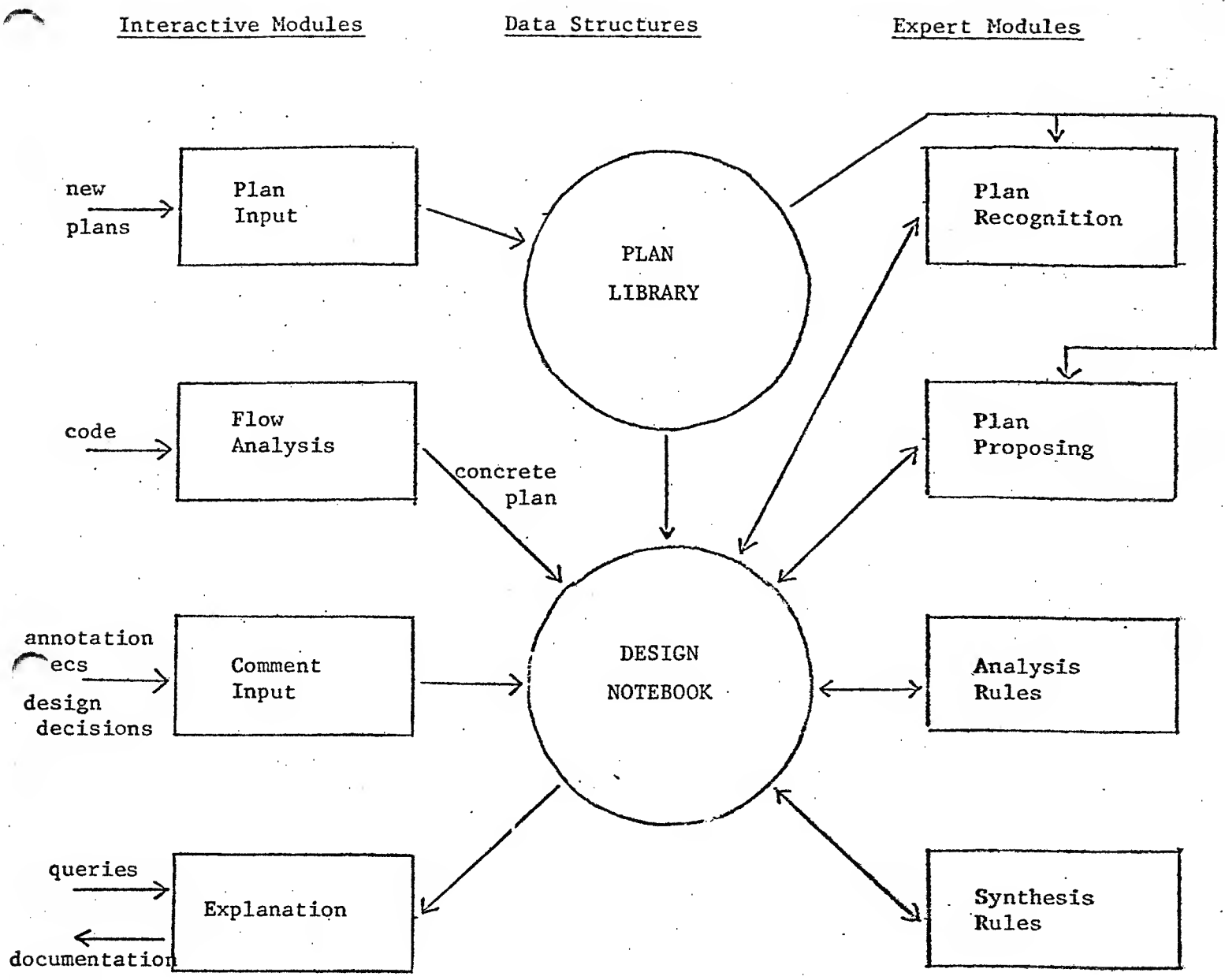


Figure 2. Structure of System.

The notebook-centered architecture allows the apprentice to generate incomplete but nevertheless useful commentary about a program design. It also allows the programmer to interleave design development with the actual coding effort in whatever order he sees fit. The apprentice gathers whatever information it is provided with and takes action whenever the information available becomes sufficient.

To get a better feeling for how the expert modules interact through the design notebook, consider what happens as the user types in the source code for a simple program (currently in LISP or FORTRAN). The main interaction in this scenario is between the plan recognizer and the analysis expert.

First the program's control flow and data flow is analyzed in order to build the most concrete plan. This becomes the current working plan for the program. The plan recognizer then examines the working plan, noting the presence of loops, recursions, and other features characteristic of certain more abstract plan types. When enough of these features combine to suggest one or more particular plan types, the recognizer records these as hypotheses in the notebook.

Hypothesizing a more abstract plan imposes various constraints. In particular, each segment of the abstract plan has an extrinsic description (a specification in terms of the role the segment plays in the abstract plan) which must be deducible from the intrinsic specifications of the matching segment(s) in the current working plan. Thus the plan recognizer is implicitly proposing theorems about the behavior of groups of segments. Control assertions requesting proofs for these theorems are added to the notebook with a justification connecting the proposed theorems to the hypothesized abstract plan.

These control assertions trigger the analysis expert, which will attempt to prove that the behavior required by the abstract plan is deducible from the known behavior of segments in the current working plan. There are three possible results.

If contradictions are discovered, a complaint is entered in the notebook. This will cause the plan recognizer to try another hypothesized plan. If no other plans are possible, an interactive module is triggered to warn the programmer that a bug is present.

If the required properties are confirmed, then the current working plan is a valid implementation of the proposed abstract plan. The apprentice has thus gone one step deeper in its understanding of the code.

It is also possible that the analysis expert will neither be able to verify the required property nor find a contradiction. One possible source of such uncertainty is the use of subroutines which have not yet been written. In such cases, the analysis expert asserts in the notebook that the desired behavior can be obtained, assuming that the unwritten subroutine can in fact be implemented. If the synthesis routines (or the programmer) should fail to do so, the validity of the original analysis is called into question.

The analysis component makes use of two closely related techniques: Symbolic Execution [Rich & Shrobe 1976] [Smith & Hewitt 1975] [Yonezawa 1976b] [Burstall 1974] and Analysis By Propagation Of Constraints [Stallman & Sussman, 1977] [Sussman & Stallman 1975]. The first step of analysis is to symbolically execute the current plan on "typical" (or symbolic) inputs. This leads to a network of situations which represent the possible intermediate and final states of the computation. Each situation consists of assertions about the symbolic objects operated on by the program.

Connecting the situations are constraint propagators which represent the behavior of plan segments. Constraint propagators can pass information both forward and backward between situations (time runs forward). To illustrate constraint propogation, consider a segment which implements the "plus" operation. If the values of the two inputs to the segment are known, the propagator creates a symbolic value in the output situation and asserts that this output is the sum of the two inputs. If one input and the output are known, propogation can assert that the other input is the difference of the output and the first input. Propogation of constraints involving quantifiers is more involved (see [Rich & Shrobe, 1976]).

Constraint propagators record in the notebook a justification for every assertion. These justifications form a network of logical dependencies between the intrinsic behaviors of sub-segments in a plan and their extrinsic descriptions, and also between the extrinsic descriptions and the behavior of the overall plan segment. Thus when plan recognition is successful, a complete teleological analysis of the program has been performed as well. This analysis is useful in many ways.

An important application is perturbation analysis. If a change is made in the specifications or implementation of a segment, a technique called Truth Maintenance [Doyle, 1977] uses the recorded dependencies to calculate exactly what information is still valid and what follows from the new assumptions, making use of what has already been learned. Thus small changes in the design may require only small amounts of new analysis. The justifications and truth maintainence system used in the apprentice maintain the code and its documentation as a unified and consistent entity which is robust under evolutionary changes. This is an important feature since large systems are currently extremely brittle in the face of change. This brittleness is due to the fact that numerous dependencies not apparent in the code underlie the correct workings of the system. Adequate written documentation and commentary is seldom supplied, and even when documentation is supplied with the initial release of a system, this documentation is infrequently maintained. Thus a programmer seeking to add features or make changes is forced to guess what the underlying assumptions are, often introducing subtle bugs in the process.

A second use of dependencies occurs when part of a proof fails. The dependencies recorded in the notebook allow the plan recognizer to analyze the failure using a technique called Dependency Directed Backtracking [Stallman & Sussman 1977] which traces backwards through the dependencies to determine which assumptions led to the contradiction. By discovering such sets of incompatible assumptions, this technique can prevent combinatorial explosions and guide the search for a correct plan more quickly.

If it turns out that the plan recognizer cannot find any abstract plan which fits the program, the teleological dependencies discovered by the analysis expert will help explain the problem to the programmer.

Although our work will concentrate more on analysis than synthesis, we can see similar mechanisms being employed in synthesis tasks as well. If a subroutine has been used in several places but has not yet been designed or coded, the constraint propagators could bring together the conditions which this segment is required to satisfy. A plan proposing expert could then search the plan library for a plan which accomplishes these or similar specifications. If such a plan is found the synthesis expert will further employ constraint propagation in an attempt to instantiate the plan in detail. By using the notebook as the communications center the system maintains the flexibility of providing only a partial solution. If it cannot reach a final synthesis on its own, the apprentice still has the option of calling on the user to guide it further. If it is convinced that the synthesis task is impossible, it can warn the programmer that he has a design bug in which he has based his program

on a subroutine which cannot be coded.

This highly flexible architecture seems best suited to the kinds of interaction we expect to be necessary for the task. For the foreseeable future we do not want to entrust the complete task of program design and synthesis to the computer. Instead we propose a cooperative environment in which both machine and programmer can analyze, debug and edit each other's code. We believe this will lead to more economical, reliable, and perspicuous programs. This environment will also serve as a workshop in which to codify and clarify the engineering knowledge which is prerequisite for the construction of the automatic programming systems of the future. Furthermore it develops these ideas within a dependency-based architecture which will permit future programming systems to be both responsible for their actions and capable of explaining the programs they have written.

## Our Research in Context

We propose to conduct this research as part of the Engineering Problem Solving Project at the MIT Artificial Intelligence Laboratory. Its goal is to uncover fundamental reasoning strategies for the design, analysis, debugging, and explanation of complex systems. Reasoning about these highly structured systems is relatively deep and requires the ability to deal with such notions as causality, teleology, and simultaneous constraints. To this end we are studying the problem solving process in electrical circuit design as well as computer programming. We are trying to capture this knowledge in the construction of computer-aided design tools for the engineer.

This endeavor has numerous antecedents. Its most direct ancestry begins with the Ph.D. theses of Sussman [Sussman 1973,1975], and Goldstein [Goldstein 1974]. These theses developed a new problem solving method we call PSBDARP (Problem Solving By Debugging Almost-Right Plans) which is based on the belief that creation and removal of bugs is an unavoidable part of solving a complex problem. Goldstein began the important task of classifying plan types. In his thesis he classified plans into three very abstract categories: round plans (loops), sequential plans, and insertion plans (formalizing the use of interrupts and state-transparent constructions).

Many of the ideas in this proposal are also a result of the general intellectual atmosphere of the MIT Artificial Intelligence Laboratory. So many of the ideas present here were developed in synergistic interactions including the authors, Aki Yonezawa and Mark Miller, Marvin Minsky and Joel Moses that it is impossible to document them individually.

Goldstein and Miller [Miller & Goldstein, 1976b], of the MIT AI Laboratory, have been studying problem solving in highly constrained domains such as the programming of a computer-controlled cursor to draw stick figures on a TV screen. They have made a catalog of very general problem solving strategies, such as decomposition and reformulation, and have organized these into an augmented transition network grammar which can be used to generate and recognize simple programs in this domain. They are also investigating the use of this grammar with novice programmers to analyze their protocols [Miller and Goldstein 1977] and to provide a highly structured programming environment [Miller and Goldstein 1976]. One emphasis of their work is on the development of a psychologically plausible theory of general problem solving processes in programming. This contrasts with and complements our effort to develop a theory which is adequate to provide computer-aided design tools for the expert programmer.

We in the Engineering Problem Solving Project have had considerable success in understanding the nature of the problem of electrical circuit design -- for an overview see [Sussman 1977]. Drew McDermott's Ph.D. thesis [McDermott 1976] developed a rule-based language, called NASL, in which it is possible to express strategies, tactics, and advice for a designer. He used this language to encode some general and specific strategies for the design of electrical circuits.

The process of localization and removal of bugs, which is part of the PSBDARP design theory, requires an approach to engineering analysis in which every result has a justification describing the exact set of assumptions it depends upon. To this end we have developed a new method of electrical circuit analysis we call Analysis by Propagation of Constraints (see [Sussman & Stallman, 1975] and [Stallman & Sussman, 1977]. We have implemented several programs which analyze circuits and can explain the basis of their deductions. The justifications constructed by Propagation of Constraints are useful both for a PSBDARP circuit designer program and also to limit the combinatorial search that occurs in analysis of a circuit. We have developed an efficient means of limiting such search which we call Dependency-Directed Backtracking [Stallman and Sussman 1977]. The use of justifications and dependencies in the control of problem-solving systems has been the subject of a recent S.M. thesis [Doyle 1977].

Allen L. Brown's Ph.D. thesis [Brown 1977] explored the use of causal and teleological reasoning in the troubleshooting of complex electrical systems. In this thesis Brown developed a set of linguistic conventions for the representation of the plan of a complex, hierarchically-structured system. Brown's methods inspired the construction of the EL analysis system [Sussman and Stallman, 1975]. Brown needed analysis by propagation of constraints to predict the consequences of a hypothesized fault in a component. These consequences are compared with the measured values as a test of the fault theory. Johan de Kleer develops this technique in his debugging program INTER [deKleer 1976]. De Kleer is now working on a Ph.D. thesis [deKleer 1977] which is aimed at a deeper theory of plans in electrical circuits and which extends Propagation of Constraints to deal with more qualitative analysis and causality.

The algebraic difficulty of determining the component values in a circuit of known topology and specifications is large. Expert circuit designers use terminal equivalence and power arguments to reduce the apparent synergy in a circuit so that their computational power can be focused. Sussman [Sussman, 1977b] introduced a new descriptive mechanism, which he calls slices, to combine the notion of equivalence with identification of parameters. Armed with appropriate slices, an automatic analysis procedure using Propagation of Constraints can be used to assign the component values in a circuit. Sussman describes techniques of formation, notation, and use of slices and how they originate in the topological design process. We believe that slices will be an important concept in program understanding.

We have also made some progress on the problems of programming. Aided by the earlier works of Brian Smith & Carl Hewitt [Hewitt and Smith 1975] and Aki Yonezawa [Yonezawa 1976b], Charles Rich and Howard E. Shrobe [Rich & Shrobe 1976] have designed and partially implemented a LISP programmer's apprentice, an interactive programming system to be used by an expert programmer in the design, implementation, and maintenance of large, complex programs. Their system is based on three forms of program description: (i) definition of structured data objects, their parts, properties, and relations between them, (ii) input-output specification of the behavior of program segments, and (iii) a hierarchical representation of the internal structure of programs (plans). Their major theoretical contribution is a representation for program plans which includes data flow, control flow, and also goal-subgoal, prerequisite, and other dependency relationships between the segments of a program. Plans are utilized in the apprentice both for describing particular programs, and also in the compilation of a knowledge base of more abstract knowledge about programming, such as the concept of a loop and its various specializations, such as search loops and enumeration loops. Rich and Shrobe also implemented a prototype

reasoning module, which can verify the correctness of plans, and a module which performs data and control flow analysis for LISP code.

Richard Waters [Waters 1976,1977] is building a system which, when completed, will be able to understand mathematical FORTRAN programs such as those in the IBM Scientific Subroutine Package. While Rich and Shrobe's work has been geared towards a general framework for program understanding, Waters has engaged in the complementary task of testing that framework within a particular problem domain. Furthermore, Waters has pushed ahead on the problem of plan recognition, using the correspondence between plan types and patterns of data and control flow to assess which plan is being used in a particular program. Waters has also characterized the teleological structure of each plan type so that the deductive part of a verification system can be given strong guidance and thereby prevented from doing much useless computation.

## Comparison with Other Work

Our research takes an engineering approach to the problem of programming. Central to this approach is the use of plans as abstractions capturing most of the knowledge commonly used by programmers. Organizing our work around a library of commonly used plans allows us to attack programming from a macroscopic viewpoint. Plans are intended to be a representation of programming knowledge that a programmer finds comfortable to use. For example, using the filtered accumulation loop plan to analyze the intersection program leads to an explanation that is very close to how many programmers themselves describe such a program. We have placed emphasis on identifying those abstractions which allow practicing program engineers to manage the complexity they have to deal with.

## Mathematics of Programs

Many researchers have worked on an attempt to reduce the problem of programming to a problem in mathematics. This work has led to development of mathematical foundations for the semantics of programs. The various semantic theories have led to the development of techniques for the verification of properties of programs and synthesis of programs.

An axiomatic semantics of programs was first developed by Robert Floyd [Floyd, 1967, 1971] and extended by C.A.R. Hoare [Hoare, 1969]. Vaughan Pratt [Pratt 1976] has recently tightened the foundations of the Floyd-Hoare system by providing it with a model. In Pratt's model programs denote binary relations on environment states. Pratt has derived interesting computability results about such logics.

The Floyd-Hoare method is to associate axioms with each primitive of a programming language which specifies how that primitive affects the state of the computation environment. For example, an assignment statement can be described by an axiom which states that after a is assigned to x, the new value of x is a; and furthermore that if P(x) is true before the assignment, then after the assignment there exists a value for which P is true, namely the old value of x. This can be written as follows:

$$P(x)\{x \leftarrow a\}\ \exists(y)P(y)\ \&\ x = a$$

This axiomatic semantics can be used in a technique for verification of properties of programs. Given a set of predicates believed to be true before a program starts execution, one can apply the axioms for each primitive in the program in turn, leading to a set of updated predicates for each exit point of the program. These are then introduced into an implication whose consequent clause is a predicate expressing those conditions which are supposed to hold after the program completes its execution. If this implication can be shown to be valid, then the program is said to be partially correct (i.e. if the program terminates, then the consequent is true).

Hoare's formulation of this method actually works in the opposite direction; he begins with predicates which express the desired terminal conditions and passes them backwards over the program primitives. This leads to an implication that the initial conditions imply some complex set of conditions resulting from passing the terminal conditions backward. Successful verification systems of this type have been described by King [1969], Deutsch [1973] and Igarashi et al. [1973]. An important extension has been made by Wegbreit [1976, 1973], who introduced a technique for generating loop invariants.

The Floyd-Hoare axiomatic tradition has spawned many descendents. Manna and Waldinger [1974] have developed a logic of programs which makes it possible to discuss termination of a program along with the Floyd-Hoare argument of partial correctness. Pratt's dynamic logic [Pratt 1976] [Litvintchouk & Pratt 1977] is claimed to subsume most existing first-order logics of programs that manipulate their environment.

An alternate semantic tradition was started by Scott [1972]. In Scott's denotational semantics, a mathematical function is constructed as the meaning of each program. Building on the work of Scott, R. Milner developed a logic for computable functions, LCF, based on the theory of the typed lambda calculus, and augmented by a powerful induction principle closely related to McCarthy's recursion induction [1963]. R. Milner and R. Weyhrauch [1972] have used LCF to formulate and prove the correctness of a small compiler.

More recently, McCarthy and Cartwright have formalized the behavior of recursive programs as formulas of first-order logic [personal communication -- Hewitt].

Our fundamental differences with the mathematical approach to the problems of programming is that it starts from the atomic elements of the domain, the programming language primitives, and then tries to relate their behavior to that of an elaborate program. Such a method is in principle possible, but the complexity of the interactions render such a direct link from language primitives to gross program behavior as intractable as a direct link from quantum physics to biochemistry. Thus we have concentrated on a hierarchy of description, using plans to describe interactions on each level.

This is not to imply that those interested in the mathematical description of programming have not recognized the pitfalls inherent in an overly microscopic approach. Indeed, C.A.R. Hoare has attempted to integrate the natural hierarchy of subroutines and data into his methods. In his axiomatic definition of the programming language PASCAL (with N. Wirth [1973]) Hoare includes a special set of axioms for procedure definition and invocation. In addition, his axioms for programs with complex data types [1972] make it possible to prove correctness properties of programs which involve complex data abstractions -- but this approach has not yet been able to reconcile the use of such high-level abstractions with the existence of mutable, recursively defined, possibly shared structures such as LISP lists. Suzuki [1976] has extended the Floyd-Hoare methodology to deal with some programs with side effects on complex data structures, but his methods lose the power of the data abstractions by construction of intractably large expressions representing the sequences of side effects.

We believe that the essence of the difficulty is the lack of a clearly defined concept of the plan of a program. The plan includes not only abstractions of the structure of the program under consideration but also commentary describing the teleology -- how the programmer maps the parts of the program to the roles they must play in the design. This information is not part of the program, but it is necessary to make effective use of the almost hierarchical structure of the program as a guide to its analysis. Basu and Misra [1975, 1976]

have made a step in this direction by identifying typical loops for which the loop invariant is already known, thus making the verification process more direct. Gerhart [Gerhart, 1975a] has also characterized typical programs for which the proof is already known. Her work relies on syntactic templates which match the surface structure of the source code, thereby achieving less generality than do plan types. But she uses program transformations to regain some of the lost generality.

J.T. Schwartz [1977] goes even further. He makes a strong argument for the inherent infeasibility of verifying the correctness of arbitrarily large programs by the flat Floyd-Hoare techniques. Instead, he is proposing to construct a library of proved root programs which represent the "fundamental and essentially indecomposable elements of algorithmic technique", and a library of proved constructors by which already proved programs can be combined to produce proved compound programs. This idea is very strong, but it enforces a strict hierarchical structure on a programmer which limits his ability to create cleverly engineered systems. Many powerful new techniques are invented through debugging of wrong but almost-right ideas. We are concerned with giving the programmer tools to help him act effectively with creativity. We do not want to provide him with a certain set of "certified" ideas out of which all other ideas must be constructed.

One problem recognized early in attempting to describe programs is that there are often several closely related ways to code the same program. For example, as demonstrated by Paterson and Hewitt [1970] and Strong [1971] recursive programs can be recoded as loops. It is clearly useful to catalogue and categorize these and similar transformations, as has been done by Darlington and Burstall [1973] and Gerhart [1975b]. We also expect to make use of transformations in understanding programs, applying them at the plan level rather than directly to the surface structure of the code in some particular programming language.

An important way our approach differs from the work reviewed above in that we are aiming at engineering analysis rather than proving of selected properties of programs. We require that our kind of analysis must produce results that a programmer can understand. Our kind of analysis must be robust in that even partially completed analysis of the kind we propose to study can be of use to an engineer, and small changes in the program plan (as may occur in normal program maintenance) should only cause small changes in the analysis.

Modern Programming Languages and Structured Programming

The problem of programming has also been attacked by building high-level languages. The idea is that some of the difficulties of programming stem from the requirement that the programmer not only specify the result to be obtained, but also the means for obtaining it in an efficient manner.

J.T. Schwartz, a pioneer of this approach, says it this way [Schwartz 1973]: "One may say that programming is optimization, and that mathematics is what programming becomes when we forget optimization and program in the manner appropriate for an infinitely fast machine with infinite amounts of memory." But the primitives of programming languages, by contrast to the primitives of the physical sciences, have no claim to fundamental status. Engineers are interested in the description of artifacts in extrinsic terms. Programming languages instead provide intrinsic forms which are extrinsically flexible: a FOR construct may be used to express both searching and enumeration plans. Others who work in programming languages are more concerned with an engineering approach in which efficiency is of the essence. The question is how one can construct a system which gives the programmer the control over the engineering tradeoffs which he needs to construct efficient programs while allowing him to think at a high level of abstraction most of the time; thus insulating him from the details of the machine, and the details of his data representation.

An important goal of research on programming language design has been the construction of languages which encourage the construction of modular programs by providing appropriate language constructs. SIMULA-67 [Birtwistle et. al. 1973] was a major breakthrough in this approach. It introduced the concept of a CLASS. This enables some of the data and procedural aspects of computational objects to be unified into a single concept.

CLU [Liskov, et. al. 1977], and ALPHARD [Wulf, et. al. 1974] have built on this idea. The CLU group has concentrated on the issues of efficiently implementing data abstractions, and on simplifying the CLASS concept. The ALPHARD group has concentrated on providing the techniques for verifying that such programs meet their specifications, building on the work of Hoare. Both CLU and ALPHARD have iteration statements that capture some of the structure of the enumeration loop plans discussed earlier in this paper. More general incremental generation structures have been developed in the stream concept of Landin; the lazy evaluation schemes of Wadsworth, Vuillemin; Morris and Henderson; Friedman and Wise; and in the sequence concept of PLASMA.

Languages for specifying the behavior of programs are the basis for communication between the implementor of a module and the users. Zilles [1975] and Guttag [1975] have developed formal descriptive system called Data Algebras for describing the behavior of immutable data abstractions. Yonezawa [1977] has developed conceptual representations for specifying objects whose behavior can change with time. Liskov and Berzins [1977] have written an excellent survey of program specification techniques.

A major idea which runs through both structured programming and automatic programming work is hierarchical stepwise refinement. As Dijkstra [1976] points out, top-down refinement is best thought of as procedural abstraction. A program can be viewed as a composition of solutions to sub-problems, each of which in turn has a similar decomposition. Thus a program can be viewed from many levels of abstraction. However, this principle has often been interpreted in an extremely rigid way which ends up proscribing certain syntactic forms, such as GOTO. This overly rigid interpretation of hierarchical refinement totally isolates the different levels and does not allow for multiple purposes of a single block of code. Yet such optimization is an important part of program engineering. In allowing mathematical elegance to take precedence over normal human engineering methodology, this interpretation of structured programming moves away from our goal of a human centered model of understanding.

## The Knowledge-Based Approach

For us the notion of hierarchy is only a starting point, a means to an end. We are more concerned with cataloguing and classifying the standard building blocks and commonly used abstractions. Knuth [Knuth, 1968] has compiled many of these standard programming techniques and given analyzes of their behavior. What we call a "plan library" is something along these lines, although we will capture a level of generality lacking in Knuth's compilation. For example, instances of the filtered accumulation loop plan appear in Knuth as many different programs. Of course, we also seek to incorporate this knowledge in a computer system which understands programs, a goal which Knuth has not set.

Early efforts to use a catalogue of specific programming knowledge include PROTOSYSTEM-1 [Ruth, 1976b,1976c], and Ruth's program analyzer [Ruth 1973, 1976a]. PROTOSYSTEM-1 is an attempt to use expert knowledge to synthesize extremely efficient file and data manipulations for business applications. It pastes together modules customized for a particular application. The IBM System 3 Application Customizer is a similar but less

ambitious venture.

Greg Ruth's program analyzer was an early attempt to capture the notion of the plan of a program. He constructed a system which analyzed correct and near-correct PL/I programs from an introductory programming class, giving specific comments about the nature of the errors detected in the incorrect programs. In Ruth's system, the class of expected programs for a given exercise is represented as a formal grammar augmented with global switches which control conditional expansions. This grammar is then used in a combination of top-down, bottom-up, and heuristic-based parsing in order to recognize particular programs.

Ruth's work has two fundamental shortcomings. First, his analysis of programs does not include any form of specification. There is no explicit statement of what a program is attempting to achieve, or what any of the subparts do individually. Thus Ruth's analysis never really captures the teleological structure of a program.

Furthermore, even if input-output specifications were given, parse trees derived from a formal grammar are inadequate for representing many important forms of teleological structure. Ruth's explanation of the purpose of a given action in a program is limited to an upward trace through the non-terminal nodes dominating the action in the program's parse tree. Tree structure is adequate to represent goal-subgoal relationships (achieve links), but does not make a crucial distinction between steps that just happen to precede each other and actual prerequisite links. Furthermore, the restriction to tree-structured analysis precludes one program action from having two different purposes; or put another way, it precludes overlapping the actions which implement distinct modules at a higher level of description.

A large body of machine-usable knowledge about LISP programming has been compiled by Green and Barstow [Green & Barstow, 1975] as part of the PSI automatic programming project [Green, 1976, 1977] at Stanford University. Their codification consists of rewrite rules that progressively refine the description of a desired program in a very high level language into a correct implementation in LISP. Furthermore, the rules in their generative grammar are annotated with pragmatic information which the PSI system uses [Barstow & Kant, 1976] to select efficient implementations from among all possible correct implementations.

As a representation of programming knowledge for use in a programmer's apprentice, the PSI rules suffer from the same two shortcomings as Ruth's work. Being developed primarily for synthesis rather than analysis or explanation, the PSI rules provide no representation of the teleological structure of the resulting program other than a history of the rule applications. Furthermore, the existing rule library includes only rules for hierarchical refinement. Program refinements which overlap module boundaries are not included in Green and Barstow's current theory.

Manna and Waldinger [1975] use a mixture of programming knowledge and mathematical formalism to perform program synthesis. They specify a program's desired behavior by input and output predicates. The intended goal is reduced by logical manipulations. They have developed additional mechanisms for the consistent handling of conjunctive goals. The synthesis process involves the formation of inductive goals which are satisfied by the construction of recursive procedures. They have written an excellent survey of synthesis techniques [Manna and Waldinger 1977].

Robert Balzer et. al. [1974] has identified the four major phases of Automatic Programming as being: Problem Acquisition, Process Transformation, Model Verification, and Automatic Coding. He proposes to investigate whether systems that implement this paradigm can be built to converse with experts [businessmen, doctors, engineers, etc.] who can <u>not</u> program to automatically produce programs in their domain of expertise. The extent to which this will be possible within the foreseeable future is unknown. We are working on a rather different problem: our goal is to construct a programers apprentice which can aid <u>expert</u> programmers in constructing large public software systems in such a way that they will be easier to write, debug, and modify. Furthermore there must be a substantial reason to believe that the programs will behave as contracted. The success of our project is not dependent on the success of the Automatic Programming projects. Indeed, it seems likely that substantial progress is necessary on the programming apprentice problem before Automatic Programming can progress past a certain point. Of course partial successes or useful techniques that are developed for automatic programming stand a good chance of being useful in our project.

## Programming System Tools

There a large number of tools currently in use to aid programmers. They include editors, debuggers, tracers, statistic gathering packages, spelling correctors and cross referencing schemes. Probably the most comprehensive and best integrated systems are the Programming Assistant of Warren Teitelman [Teitelman 1977] and the M.I.T. LISP machine [Greenblatt et. al. 1977]. Currently such systems have no knowledge of the plans, specifications, or teleology of programs and thus are severely limited in the aid they can give to the programmers. We propose to remedy this deficiency by the development of a programmer's apprentice.

## Acknowledgments

We would like to thank Mark Miller and Marvin Minsky for their extensive help in the development of this document. Discussions with Jon Doyle, Johan De Kleer, Guy Steele, Drew McDermott and Marilyn Matz were critical to solidifying our ideas. Ira Goldstein, Mike Dertouzos, and Patrick Winston also made useful suggestions.

## Bibliography

Balzer, R. 1973 "Automatic Programming", Institute Technical Memo, University of Southern California / Information Sciences Institute, Los Angeles, Cal.

Balzer, R. et. al. 1974 Domain-Independent Automatic Programming, ISI/RR-73-14 University of Southern California (March 1974).

Barstow, David 1977, Automatic Construction of Algorithms and Data Structures, PhD. Thesis, Stanford University, September 1977.

Barstow, David and Kant, Elaine 1976, "Observations on The Interaction of Coding and Efficiency Knowledge in the PSI Program Synthesis System", Proceedings of The Second International Conference on Software Engineering, San Francisco, California, October 1976, pp. 19 - 31.

Basu, S. & Misra, J. 1975, "Proving Loop Programs", IEEE Trans. on Software Engineering Vol. 1 Number 1, March 1975.

Basu, S. & Misra, J. 1976, "Some Classes of Naturally Provable Programs", Second International Conference on Software Engineering, pp. 400-406, Oct. 1976.

Bauer, M. 1975 "A Basis for the Acquisition of Procedures from Protocols", Fourth International Joint Conf: on A.I., U.S.S.R.

Birtwistle, G.M.; Dahl, Ole-Johan; Myhrhaug, B.; and Nygaard, K. 1973 SIMULA BEGIN. Auerbach. 1973

Boyer, R.S. & Moore, J.S. 1975 "Proving Theorems About LISP Functions", JACM vol. 22 no. 1, January 1975.

Brown, A.L. 1977 Qualitative Knowledge, Causal Reasoning, and the Localization of Failures, M.I.T. Artificial Intelligence Laboratory Technical Report 362, March 1977.

Burstall, R. M. 1974, "Program Proving As Hand Simulation and A Little Induction", Proceedings of IFIP Conference 1974.

Burstall, R. M. 1969 "Proving Properties of Programs by Structural Induction", Comput. Jl. vol. 12, pp. 4-8

Burstall, R. M. 1972 "Some Techniques for Proving Properties of Programs Which Alter Data Structures", Machine Intelligence 7, Edinburgh University Press.

Dahl, O.J., Dijkstra, E., And Hoare, C. A. R. 1972 Structured Programming, Academic Press, 1972.

Darlington, J. and Burstall, R.M. 1973 "A System Which Automatically Improves Programs", Third International Joint Conf. on A.I., Stanford U.

Darlington, Jared L. 1973a "Automatic Program Synthesis in Second-Order Logic", Third International Joint Conf. on A.I., Stanford U.

Deutsch, L.P. 1973, An Interactive Program Verifier, PhD. Thesis University of California at Berkeley, June 1973.

Dijkstra, E.W. 1976, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J. 1976

DeKleer, Johan 1976, "Local Methods for Localization of Faults in Electronic Circuits", M.I.T. Artificial Intelligence Laboratory Memo 394.

DeKleer, Johan 1977, "A Theory of Plans for Electronic Circuits", MIT Artificial Intelligence Laboratory Working Paper 144, May 1977.

DeKleer, J., Doyle, J., Steele, G. & Sussman, G.J. "AMORD: Explicit Control of Reasoning", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, August 1977.

Donzea-Gouge, V., Huet G., Kahn, G., Lang, B., and Levy, J.J. 1975 "A Structure-Oriented Program Editor: A First Step Towards Computer Assisted Programming", Report 114, Institut de Recherche en Informatique et Automatique, France.

Doyle, Jon 1977 Truth Maintenance Systems for Problem Solving, M.I.T. Artificial Intelligence Laboratory M.S. Thesis May 1977. Also to appear as M.I.T. Artificial Intelligence Laboratory Technical Report 419, 1977.

Floyd, R. W. 1967 "Assigning Meaning to Programs", Mathematical Aspects of Computer Science. J.T. Schwartz (ed.) vol. 19, Am. Math. Soc. pp. 19-32. Providence Rhode Island.

Floyd, R.W. 1971 "Toward Interactive Design of Correct Programs", IFIP, 1971.

Gerhart, S. L. 1975a, "Knowledge About Programs: A Model and Case Study", SIGPLAN Notices, Vol. 10, Num. 6, Proceedings of the International Conference on Reliable Software.

Gerhart, S.L. 1975b "Correctness-Preserving Program Transformations", Proc. of 2nd Symp. on Principles of Programming Languages, Palo Alto.

German, S. & Wegbreit, B. 1975, "A Synthesizer of Inductive Assertions", IEEE Transactions on Software Engineering Vol. 1 Num. 1, March 1975.

Goldstein, Ira 1974, Understanding Simple Picture Programs, MIT Artificial Intelligence Laboratory Technical Report 294, September 1974

Goldstein, Ira 1976, "The Computer As Coach", MIT Artificial Intelligence Laboratory Memo 389, December 1976.

Goldstein, I.P. and Miller, M.L. 1976 "Structured Planning and Debugging, A Linguistic Theory of Design", MIT AI Lab Memo 387. December, 1976.

Green, G.C. 1976, "The Design of The PSI Program Synthesis System", Proceedings of The Second International Conference on Software Engineering, San Francisco, October 1976, pp. 4 - 18.

Green, G.C. 1977, "A Summary of The PSI Program Synthesis System", Proceedings of The Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977, pp. 380 - 381.

Green, G.C. & Barstow, D.R. "Some Rules for the Automatic Synthesis of Programs", IJCAI-4 Tbilisi, USSR, September 1975.

Guttag, J. V. 1975, "The Specification and Application to Programming of Abstract Data Types" Technical Report CSRG-59. University of Toronto. September, 1975.

Hewitt, Carl 1972, Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language For Proving Theorems and Manipulating Models in a Robot, M.I.T. Artificial Intelligence Laboratory Technical Report TR-258, April 1972.

Hewitt, C. & Smith B.C. 1975, "Towards A Programming Apprentice", IEEE Transactions on Software Engineering, Vol. SE-1 No. 1, March 1975.

Hoare, C.A.R 1969, "An Axiomatic Basis for Computer Programming", Comm. ACM, vol. 12, number 10, October 1969, pp. 576-580,583.

Hoare, C.A.R 1971, "Proof of A Program: Find", Comm. ACM, vol. 14, number 1, January 1971, pp. 39-45.

Hoare, C.A.R. 1972 "Proof of Correctness of Data Representations", Acta Informatica, 1,4, pp. 271-281.

Hoare, C.A.R. and Wirth, N. 1973 "An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica, 2,4, pp. 335-355.

Igarashi S., London R., and Luckham D. 1973, Automatic Program Verification I: A Logical Basis and Its Implementation, Stanford AIM-200, May 1973.

Kant, E. "The Selection of Efficient Implementations for A High Level Language", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, August 1977.

Katz, S.M., and Manna, Z. 1973, "A Heuristic Approach to Program Verification", Third International Joint Conf. on A.I., Stanford U.

Katz, S. & Manna, Z. 1976, "Logical Analysis of Programs", Communications of the ACM, Vol. 19 Num. 4 pp.188-206 April 1976.

King, J. 1969, A Program Verifier, Carnegie Mellon University, 1969.

King, J.C. 1971 "Proving Programs to be Correct", IEEE Trans. on Computers, C-20, 11, Nov. 1971.

King, J.C. 1976) "Symbolic Execution and Program Testing", Comm. of the ACM, July, Vol.

19, No. 7, p. 385.

Knuth, D.E. 1968 The Art of Computer Programming, Vol. I, Addison-Wesely.

Liskov, B. 1974, "A Note on CLU", MIT/Computation Structures Group Memo 112, MIT/LCS, November 1974.

Liskov, B.; Snyder, Alan; Atkinson, Russell; and Schaffert, Craig; 1977, "Abstraction Mechanisms in CLU", Communications of the ACM, August 1977, pp. 564 - 576.

Liskov, B. and Berzins, V. "An Appraisal of Program Specifications" M.I.T. Computation Structures Group Memo 141-1. April 1977.

Liskov, B. and Zilles S. 1974 "Programming with Abstract Data Types", Proc. of Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4.

Liskov, B. & Zilles, S.N. 1975, "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, Vol. SE-1 No. 1, March 1975.

Litvintchouk, S.D. and Pratt, V.R. 1977, A Proof-Checker for Dynamic Logic, MIT AI Memo 429, June 1977.

London, R. 1975 "A View of Program Verification", ACM SIGPLAN Notices, Vol. 10, No. 6, Proc. of International Conf. on Reliable Software.

Long, W.J. 1977 "A Program Writer". MIT/LCS/TR-107, November 1977 (Ph.D. Thesis)

Manna, Z. and Pnueli, A. 1974, "Axiomatic Approach to Total Correctness of Programs", in Acta Informatica 3, pp. 253-263, 1974.

Manna, Z. and Waldinger, R. 1975, "Knowledge and Reasoning in Program Synthesis", Artificial Intelligence 6, pp. 175-208.

Manna, Z. and Waldinger, R. 1976, "Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness" in Proc. 2nd Int. Conf. on Software Engineering, October 1976.

Manna, Z. and Waldinger, R. 1977, Synthesis: Dreams => Programs, Stanford Research

Institute Technical Note 156, November 1977.

McCarthy, J. 1963, "Towards a mathematical science of computation" Proc. IFIP Congress 62, pp. 21-28, Amsterdam: North Holland.

McCarthy, J. and Hayes, P. 1969 "Some Philosophical Problems from the Standpoint of Artificial Intelligence", Machine Intelligence 4, American Elsevier, N.Y.

McDermott, Drew Vincent 1976, "Flexibility and Efficiency in a Computer Program for Designing Circuits", MIT PhD. Thesis, September 1976.

Mikelsons, M. 1975 "Computer Assisted Application Definition", Proc. of 2nd ACM Symp. of Principles of Programmings Languages, Palo Alto.

Miller, M. L. and Goldstein, I.P. 1976a "SPADE: A Grammar Based Editor For Planning and Debugging Programs", MIT AI Lab Memo 386. December 1976.

Miller, M. L. and Goldstein, I.P. 1977a. "Structured Planning and Debugging" IJCAI-77. August 1977.

Miller, M. L. and Goldstein, I.P. 1977b "Problem Solving Grammars as Formal Tools for Intelligent CAI" ACM77. October, 1977.

Minsky, M., "Form and Content in Computer Science", JACM 17, No. 2, April 1970, pp. 197-215, 1970 ACM Turning Lecture.

Moore, J.S. 1974, "Introducing PROG into the PURE LISP Theorem Prover", Xerox PARC Report CSL-74-3.

Moore, Robert Carter 1975, Reasoning From Incomplete Knowledge In A Procedural Deduction System, MIT/AI-TR-347 December 1975.

Morris, J.H. & Wegbreit, B. 1977, "Subgoal Induction", Communications of the ACM, Vol. 290 Num. 4 pp. 209-222, April 1977.

Parnas, D.L. 1972, "A Technique for Software Module Specification with Examples", CACM Vol. 15, No. 5.

Paterson, M. and Hewitt, C. 1970 "Comparative Schematology", Conference Record, ACM Conference on Concurrent Systems and Parallel Computation (1970).

Pratt, V. 1976, "Semantical Considerations on Floyd-Hoare Logic", MIT/LCS/TR-168, September 1976; also in Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci., pp. 109-121, 1976.

Rich, C. 1977, Plan Recognition In A Programmer's Apprentice", MIT/AI Working Paper 147, May 1977.

Rich C. and Shrobe H. 1976, An Initial Report On A LISP Programmer's Apprentice, MIT/AI/TR-354, December 1976.

Ruth, Gregory 1973, Analysis of Algorithm Implementations M.I.T. Ph.d. Thesis, Project MAC Technical Report 130.

Ruth, Gregory 1976a, "Intelligent Program Analysis", Artificial Intelligence 7, Spring 1976, pp. 65 - 85.

Ruth, Gregory 1976b, "Protosystem I: An Automatic Programming System Prototype", TM-72, MIT Laboratory for Computer Science, 1976 (also to appear in the proceedings of the 1978 NCC in abbreviated form).

Ruth, Gregory. 1976c, "Automatic Design of Data Processing Systems", 23rd ACM Symposium on Principles of Programming Languages, 1976.

Sacerdoti, Earl D. 1975, "The Non-Linear Nature of Plans" Stanford Research Institute A.I. Group Technical Note 101.

Sacerdoti, Earl D. 1975a, "A Structure for Plans and Behavior", SRI Technical Note 109.

Schwartz, J.T. 1973, On Programming, An Interim Report on the SETL Project; Installment I: Generalities, Courant Institute of Mathematical Sciences, New York University, February 1973.

Schwartz, J.T. 1977, "On Correct Program Technology", in Courant Computer Science Report #12, September 1977.

Scott, D. 1972 "Lattice Theory, Data Types and Semantics", in Formal Semantics of Programming Languages, Rustin ed, Prentice-Hall, p. 65.

Shaw, D., Swartout, W., and Green, C. 1975 "Inferring LISP Programs from Examples", Fourth International Joint Conf. on A.I., U.S.S.R.

Shrobe, Howard E. 1978, "Plan Verification in A Programmer's Apprentice", M.I.T. Artificial Intelligence Laboratory Working Paper * 158, January 1978.

Siklossky, L. and Sykes D. 1975 "Automatic Program Synthesis from Example Problems", Fourth International Joint Conf. on A.I., U.S.S.R.

Spitzen, J. & Wegbreit, B. 1975, "The Verification and Synthesis of Data Structures", Acta Informatica 4, 1975.

Stallman, Richard and Sussman, G. J. 1977, "Forward Reasoning and Dependency-Directed Backtracking In A System for Computer-Aided Circuit Analysis", Artificial Intelligence Journal, October 1977.

Steele, Guy L. 1977, "Debunking the 'Expensive Procedure Call' Myth", Proceedings of ACM-77, October 1977.

Strong, H.R. 1970, Translating Recursion Equations into Flow Charts, Reports RC 2834 (March 1970) and RC 2859 (April 1970), IBM Yorktown Heights.

Sussman, G.J. 1973, A Computational Model of Skill Acquisition, M.I.T. Department of Mathematics Ph.D. Thesis; M.I.T. Artificial Intelligence Laboratory Technical Report 297, August 1973; A Computer Model of Skill Acquisition, New York, American Elsiver 1975.

Sussman, G.J. 1977a, "Electrical Design: A Problem for Artificial Intelligence Research", Proceedings of The Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977.

Sussman, G. J. 1977b, "SLICES: At The Boundary Between Analysis and Synthesis", M.I.T. Artificial Intelligence Laboratory Memo 433, July 1977. (Also to appear in The Proceedings of The IFIP Working Conference on Artificial Intelligence and Pattern Recognition in Computer Aided Design in 1978.)

Sussman, G. J. and Stallman Richard 1975, "Heuristic Techniques in Computer Aided Circuit Analysis", IEEE Transactions on Circuits and Systems, Vol. CAS-22, No. 11, November 1975.

Suzuki, N. 1976, Automatic Verification of Programs with Complex Data Structures, Stanford AIM-279, February 1976.

Teitelman, W. 1977 "A Display Oriented Programmer's Assistant" IJCAI-77. August 1977.

Waldinger, R. and Levitt, K.N. 1974 "Reasoning About Programs", Artificial Intelligence 5, pp. 235-316.

Waldinger, Richard 1975, "Achieving Several Goals Simultaneously" Stanford Research Institute A.I. Group Technical Note 107.

Waters, R.C. 1976, "A System for Understanding Mathematical FORTRAN Programs", M.I.T. Artificial Intelligence Laboratory Memo 368, August 1976.

Waters, R.C. 1977, "A Method, Based on Plans, for Understanding How a Loop Implements a Computation", M.I.T. Artificial Intelligence Laboratory Working Paper 150, July 1977.

Wegbreit, B. 1973, "Heuristic Methods for Mechanically Deriving Inductive Assertions", Third International Joint Conf. on A.I., Stanford U.

Wegbreit, B. 1974, "The Synthesis of Loop Predicates", Communications of The ACM, Vol. 17, pp. 102-112, Feb. 1974.

Wegbreit, B. 1976, "Constructive Methods In Program Verification", Xerox Palo Alto Research Center CSL-76-2, July 1976.

Wilczynski, D. 1975 "A Process Elaboration Formalism for Writing and Analyzing Programs", U. of S. Cal. Information Sciences Inst., ISI/RR-75-35.

Winograd, Terry 1973 "Breaking the Complexity Barrier Again" Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973.

Wulf, W.A. 1974, "ALPHARD: Towards a Language to Support Structured Programming",

Carnegie Mellon University Dept. of Comp. Sci., April 1974.

Yonezawa, A. 1975, "Meta-Evaluation of Actors With Side Effects", MIT/AI Working Paper 101, June 1975.

Yonezawa, Aki 1976a "Meta-Evaluation for Verification and Analysis of Actor Programs", Draft paper, M.I.T. A.I. Lab.

Yonezawa, A. 1976a, "Symbolic-Evaluation As An Aid To Program Synthesis", MIT/AI Working Paper 124, April 1976.

Yonezawa, A. 1976b, "Symbolic Evaluation Using Conceptual Representations For Programs With Side-Effects", MIT/AI Memo 399, December 1976.

Yonezawa, A. 1977, "Verification and specification Techniques for Parallel Programs baed on Message-Pasing Semantics". M.I.T. PhD. December, 1977.

Zilles, S. 1975, "Abstract Specification for Data Types", IBM Research Laboratory, San Jose California, 1975.